http://dx.doi.org/10.5755/j01.eie.25.6.24824 ELEKTRONIKA IR ELEKTROTECHNIKA, ISSN 1392-1215, VOL. 25, NO. 6, 2019

A Hardware Approach of a Low-Power IoT Communication Interface by NXP FlexIO Module

Libor Chrastecky, Jaromir Konecny, Martin Stankus, Michal Prauzek*
Faculty of Electrical Engineering and Computer Science, VSB Technical University of Ostrava, Ostrava, Czech Republic
michal.prauzek@vsb.cz

***Abstract*—This article describes implementation possibilities of specialized microcontroller peripherals, as hardware solution for Internet of Things (IoT) low-power communication, interfaces. In this contribution, authors use the NXP FlexIO periphery. Meanwhile, RFC1662 is used as a reference communication standard. Implementation of RFC1662 is performed by software and hardware approaches. The total power consumption is measured during experiments. In the result section, authors evaluate a time-consumption trade-off between the software approach running in Central Processing Unit (CPU) and hardware implementation using NXP FlexIO periphery. The results confirm that the hardware-based approach is effective in terms of power consumption. This method is applicable in IoT embedded devices.**

***Index Terms*—Energy harvesting; Low-power electronics; Finite state machine; FlexIO.**

I. INTRODUCTION

The majority of today’s Internet of Things (IoT) devices are designed as low-cost and low-power embedded platforms [1]. Generally, current IoT design goals require modern approaches that can achieve minimal power consumption and minimal processing times [2]. This research direction is very important, especially in battery-powered or harvesting IoT platforms [3].

Low-power devices usually operate in duty-cycle scenario [4] when run and sleep modes are changing in regular or adaptive intervals [5]. There are two possible approaches to obtain a power reduction. The first method is based on an absolute power consumption reduction in run and sleep modes [6]. The second method aims for effective computing algorithms [7] that allow minimization of time spent in run modes, which in turn decreases total power consumption.

The goal of this work is to introduce a method for the hardware-based approach of energy demanding communication protocols to improve low-power designs in run modes of IoT devices. This contribution presents utilization possibilities of a special microcontroller (MCU) module to reduce processing times in low-power communication protocols implementation. The NXP FlexIO module demonstrates a hardware-based approach communication protocol implementation represented by RFC1662 standard, which can be used for IoT embedded devices. A transfer of computational tasks from Central Processing Unit (CPU) to special peripherals, such as FlexIO, causes an increase of instant supply current demands in the run mode [8], [9]. Therefore, a trade-off between the task duration and power supply demands must be examined to be able to select a proper approach for the target application.

This paper is organized into five sections. Introduction provides a short overview of the current state of the art in IoT application and applications of special hardware modules. The NXP FlexIO periphery and selected communication standard RFC1662 are described in Section II. The software and hardware implementations of the proposed approach are detailed in Section III. The comparison between the software and hardware implementations is evaluated in Section IV. The final section (Section V) brings major conclusions and outlines directions for the future research.

II. BACKGROUND

This section introduces two technologies implemented in the experimental part of this work, which are NXP FlexIO module and communication standard RFC1662.

A. *NXP FlexIO Module*

NXP FlexIO is used as the communication MCU module. It can emulate various protocols for serial and parallel communication, such as UART, SPI or I^2C. The module consists of three main parts (Fig. 1), which are shifters, timers, and pins. The input data are uploaded to the shifter and, then, shifted to the output pin by the clock generated by the timer.

FlexIO can be used for various use-cases, such as emulation of a serial or parallel communication interfaces, user-defined time charts and trigger signals generation, creation of output logical function through logical look-up

Manuscript received 7 February, 2019; accepted 5 September, 2019.
This work was supported by the project “Development of algorithms and systems for control, measurement and safety applications V” (No. SP2019/107) of Student Grant System, VSB-TU Ostrava. This work was also supported by the European Regional Development Fund in the Research Centre of Advanced Mechatronic Systems project (No. CZ.02.1.01/0.0/0.0/16_019/0000867) within the Operational Programme Research, Development and Education.

tables or create a programmable hardware Finite State Machine (FSM). A programmable FSM allows for replacement of a system control performed by a central processing unit (CPU).

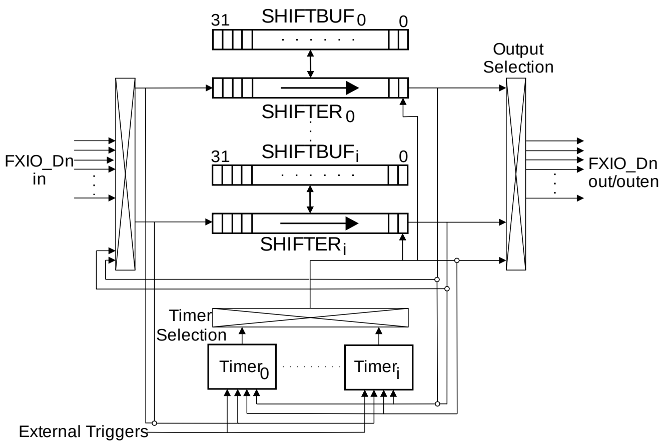

Fig. 1. Block diagram of NXP FlexIO module.

Figure 1 shows the implementation of NXP FlexIO module on NXP Kinetis ARM Cortex-M KL28Z MCU [10]. FlexIO consists of 16-bit counter with trigger signal support, reset, and start and stop conditions. It includes program logic blocks that allow for implementation of digital logic functions on a chip and adjustable interactions possibilities among internal and external peripherals.

FlexIO features include:

– 32-bit shifter registers with transmission, receive, and data comparison mode,

– Double cache for shifting operations during data transfer,

– Internal shifter with chain support to large data transfer,

– 1, 2, 4, 8, 16 or 32 multi-bits shifting width support for parallel interfaces,

– Programmable FSM allowing the transfer of basic system control function from a CPU with up to 8 status support, 8 outputs, and 3 status selector inputs.

B. Communication Standard RFC1662

For FlexIO module testing purposes, byte-oriented point-to-point communication standard RFC1662 [11] is used. In this standard, data are assembled as a frame, which starts and ends with a specific flag. When the flag is found in the data, it must be replaced by two-byte escape sequence (first ESC, second ESC_FLAG). If ESC itself appears in the data, it must be also replaced by another escape sequence (first ESC, second ESC_ESC).

Data to send

Data 0x2A	Data 0x8E	Data 0x7E	Data 0x45	Data 0x7D	Data 0x55

Escaped special chars

Data 0x2A	Data 0x8E	ESC 0x7D	ESC_FLAG 0x5E	Data 0x45	ESC 0x7D	ESC_ESC 0x5D	Data 0x55

Packet to send

FLAG 0x7E	Data 0x2A	Data 0x8E	ESC 0x7D	ESC_FLAG 0x5E	Data 0x45	ESC 0x7D	ESC_ESC 0x5D	Data 0x55	FLAG 0x7E

Fig. 2. RFC1662 frame schematic.

The RFC1662 standard defines the flag value as 0x7E. The ESC value is defined as 0x7D, ESC_FLAG and ESC_ESC are defined as 0x5E, and 0x5D, respectively. Figure 2 shows an example of the frame processing (escaping).

The data to send contain flag (0x7E) and ESC (0x7D) byte. Data to send must be escaped and the flag and ESC are replaced by the escape sequence. Then, the start and terminate flags are added to the escaped data and this comprises a complete frame to be sent.

III. Implementation and Testing

This section brings an overview of FSM implementation in terms of software and hardware approach. In testing subsection, the parameters of testing procedure are presented.

A. Implementation

In the experimental section, we designed FSM to compare the software implementation with the hardware approach using FlexIO module. The FSM implementation represents RFC1662 protocol and the goal of this experiment is the evaluation in term of energy demands.

The FSM implementation using the data link layer RS-486 is depicted in Fig. 3.

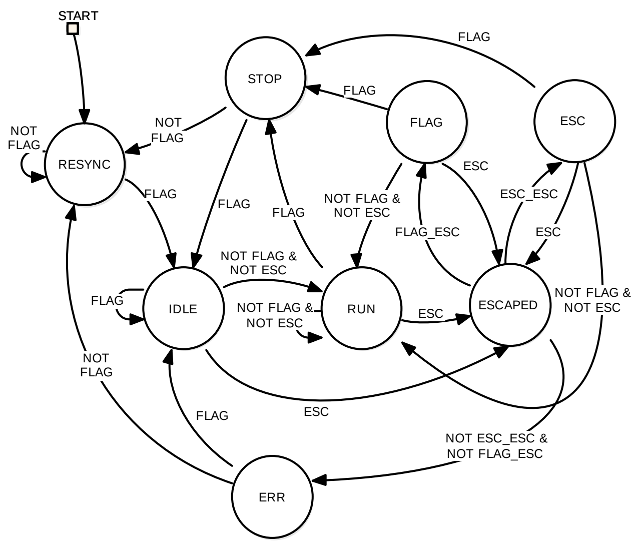

Fig. 3. Design of finite state machine representing RFC1662 functionality.

A device implementing FSM starts in RESYNC state, because communication might already run and unchecked start may cause a communication error. In RESYNC state, FSM expects the FLAG, i.e., start or end of the message. After receiving the FLAG, the FSM changes the state into IDLE. If FSM receives a FLAG again, the state will be not changed and the start of the message is indicated again. If FSM receives (in IDLE) other character than FLAG, the FSM changes the state to RUN state and this state is dedicated to the message content reception. If FLAG character is received in RUN state, the FSM goes to STOP state and entire frame is received.

If FSM receives ESC character in RUN, IDLE, ESC or FLAG state, FSM expects an escape sequence and active state is changed to ESCAPED state. The ESCAPED state means that one of two special characters (ESC_ESC and FLAG_ESC) is expected. If FSM receives ESC_ESC or

ESC_FLAG, the FSM enters ESC or FLAG state, respectively, and corresponding value is written to the receiving data buffer. If FSM receives other character than ESC_ESC or ESC_FLAG in ESCAPED state, the message is corrupted and FSM goes to ERR state, and waits for the FLAG character. The FLAG character can also be received in FLAG or ESC state and indicates the end of the received frame. So, FSM goes to STOP state.

Software implementation uses the Universal Asynchronous Receiver Transmitter (UART) module as a physical layer. It is depicted in Fig. 4. FSM implementation is coded in C language and it is running in CPU ARM Cortex-M core. Each received byte causes an interrupt and received characters are read and processed by C-coded algorithm.

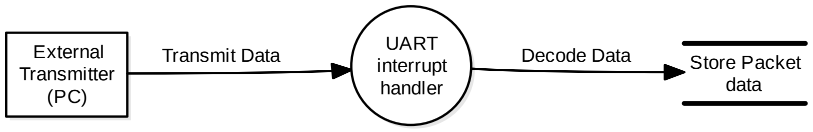

Fig. 4. Basic scenario of software finite state machine design.

Hardware implementation also uses UART as the physical layer (Fig. 5). The FSM is implemented using FlexIO module. Each state is denoted as a 3-bit value, i.e., 0–7 and these values are stored in the look-up table LUT Output. The process starts by receiving one byte by UART and the received byte and current FlexIO state together form an input address of the look-up table, where the next FlexIO state value is located. The value of the next state is written to the FlexIO input and, then, the FlexIO timer trigger is activated by Trigger MUX Control (TRGMUX). This event will cause the state changing.

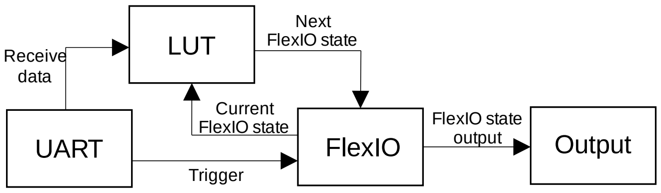

Fig. 5. Functional diagram of hardware approach using FlexIO.

Each state processes a specific action, which is defined by FSM. Each action launches a Direct Memory Access (DMA) channel and DMA engine performs target action (e.g., constant moved to the defined memory space). DMA events are represented in Fig. 5 by the action arrows.

In Fig. 6, we can see the DMA channel functionality with the scatter/gather operation, which allows for handling of multiple transmissions by loading new Transfer Control Descriptor (TCD) structure. TCD structure is implemented in DMA channel configuration and affects the transmission parameters. Upon each transfer via DMA channel a new TCD structure is written to DMA transfer control registers, so that allows for modification of source and destination addresses for each transfer.

When UART module receives one byte, it causes DMA request. DMA transfers UART receive register to a data storage variable and UART receiving register content is automatically erased. Then, a current state value and UART variable value are transferred to a source address in TCD structure, which comprises an address of an item in LUT_NextState. The item in LUT_NextState represents an address of the second look-up table item in LUT_Output and it is transferred to a source address in a new TCD structure.

LUT_Output is an array consisting of next state values. DMA engine transfers the LUT_Output item value to the MCU output pins. Output pins are directly connected to FlexIO input pins. Therefore, we need a trigger that handles the input FlexIO pin changed event. First, we create special trigger on a MCU pin, which is internally connected with FlexIO Timer by TRGMUX. FlexIO Timer starts counting and stops immediately. After FlexIO Timer is stopped, the FlexIO input values are read, and these values represent the new applied FSM state.

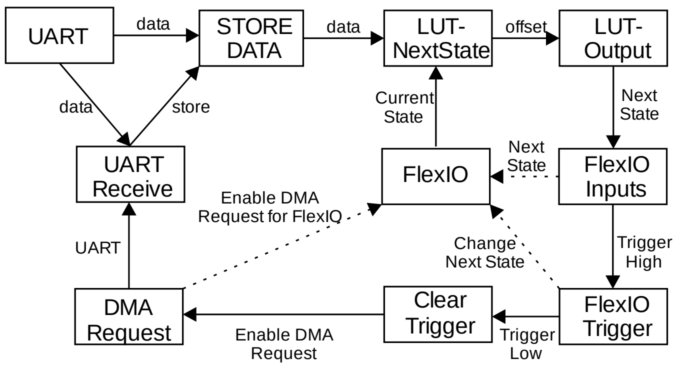

Fig. 6. Direct memory access engine using FlexIO module.

B. Testing Procedure

For evaluation purposes, a testing application in C# language was designed. The application allows to send data to the testing setup, while the test itself includes the evaluation of frame reception with different sizes, ESC characters amount, and errors count. The test aims for a comparison of hardware and software approaches in terms of processing duration and total energy demands.

Table I shows an overview of the experimental testing parameters. The transmitting speed in UART peripheral is set to 115200 Baud and the maximum packet size is 1500 bytes. The test is designed to evaluate responses to the frames without any error and the frames with one error.

TABLE I. TESTING MESSAGE PARAMETERS.

Testing	Parameters
Message length	500/1000/1500
Number of ESC character within message	0/10/20
Number of Errors within message	0/1

IV. RESULTS

Firstly, the power consumption of hardware and software approaches was evaluated. The testing setup operated at 3.3 V and the current in active state was 15.52 mA for the software implementation and 16.59 mA for hardware implementation. Both values were measured by picoAmmeter Keithley 6485. The hardware approach using FlexIO module has higher power consumption because this solution uses more hardware peripherals (e.g., DMA engine) than software approach.

Total processing times of hardware and software approaches (obtained by DSOX2024A Oscilloscope) are presented in Table II. The results include frame receiving duration in software and hardware implementation and energy saving by FlexIO approach for each testing frame described by the amount of characters in frames. In total, the average transmission time for software implementation is 136.4 ms and for solution using FlexIO – 80.3 ms. The hardware approach achieves lower processing time because this solution does not need CPU operation and uses mostly hardware modules, such as DMA, FlexIO or TRGMUX.

Figure 7 shows differences between the total power consumption of hardware and software approaches. In all cases, the hardware approach has lower energy demands even with higher instant power consumption. On average, the hardware implementation consumption is approximately 37 % lower than software approach. However, the disadvantage of hardware implementation is that it uses significantly more memory space than software approach due to fact that implementation needs to encode the received characters through look-up tables to 3-bit values for the FlexIO module.

TABLE II. RESULTS OF ALGORITHM PROCESSING TIMES.

Packet	Char	SW time [ms]	HW time [ms]	Energy (HW-SW) [%]
500-0-0	498	110.8	49	54
500-0-1	199	73.6	19	72
500-10-0	488	119	48	57
500-10-1	365	85.2	36	55
500-20-0	478	114	48	55
500-20-1	394	105.2	40	60
1000-0-0	998	146	95	30
1000-0-1	399	106	39	61
1000-10-0	988	144	95	29
1000-10-1	413	111.6	42	62
1000-20-0	978	148	95	31
1000-20-1	809	129.2	79	35
1500-0-0	1498	176	143	13
1500-0-1	649	112.4	62	41
1500-10-0	1488	193	144	20
1500-10-1	1311	182	128	26
1500-20-0	1478	208	143	27
1500-20-1	1454	191	141	21
Average	826.9	136.4	80.3	37

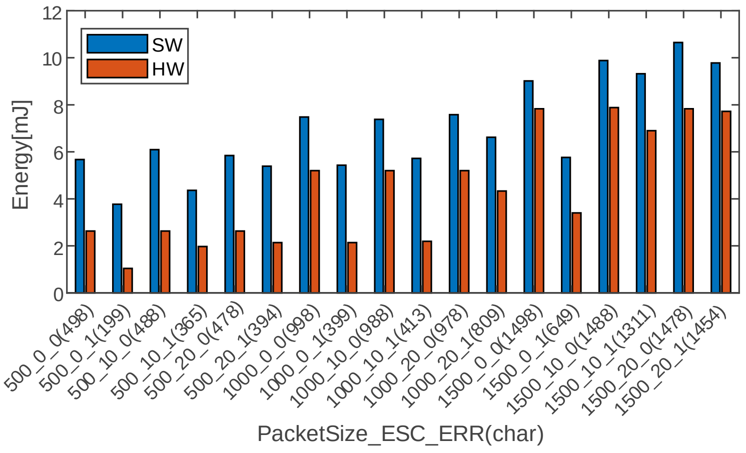

Fig. 7. Bar graph comparison between energy consumption in hardware and software implementation.

V. CONCLUSIONS

This article introduces low-power IoT communication interface featuring FlexIO module in communication protocol FSM implementation for standard byte-oriented point-to-point serial communication RFC1662. This approach represents hardware-based method to save energy in IoT communication scenarios. The novel method is compared with reference software FSM implementation to determine the energy saving. FlexIO approach achieves better results in terms of processing times and total energy efficiency. On average, the hardware approach is 37 % more effective in terms of energy consumption and 41 % in term of the processing time. Also, the implementation of the FSM via the FlexIO periphery allows transfer of computational power from CPU to HW modules and CPU computational power can be used to perform another task.

In the future, the FSM realized by FlexIO could be used to analyse inserted sync flags, or to distinguish odd and even frames in interleaved video signal. As the most promising area of interest, we are considering implementation of this method in battery-powered or harvesting IoT embedded platforms, where lower energy demands could extend operational times.

CONFLICTS OF INTEREST

The authors declare that they have no conflicts of interest.

REFERENCES

[1] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac, "Internet of Things: Vision, applications and research challenges", *Ad Hoc Networks*, vol. 10, no. 7, pp. 1497–1516, 2012. DOI: 10.1016/j.adhoc.2012.02.016.

[2] K. Georgiou, S. Xavier-De-Souza, and K. Eder, "The IoT energy challenge: A software perspective", *IEEE Embedded Systems Letters*, vol. 10, no. 3, pp. 53–56, 2018. DOI: 10.1109/LES.2017.2741419.

[3] A. Raj and D. Steingart, "Review – power sources for the Internet of Things", *Journal of the Electrochemical Society*, vol. 165, pp. B3130–B3136, 2018. DOI: 10.1149/2.0181808jes.

[4] C. Vigorito, D. Ganesan, and A. Barto, "Adaptive control of duty cycling in energy-harvesting wireless sensor networks", in *Proc. of 2007 4th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks*, SECON, 2007, pp. 21–30. DOI: 10.1109/SAHCN.2007.4292814.

[5] P. Musilek, M. Prauzek, P. Kromer, J. Rodway, and T. Barton, "Intelligent energy management for environmental monitoring systems", *Smart Sensors Networks: Communication Technologies and Intelligent Applications, Intelligent Data-Centric Systems*, pp. 67–94, 2017. DOI: 10.1016/B978-0-12-809859-2.00005-X.

[6] M. Prauzek, P. Musilek, and A. Watts, "Fuzzy algorithm for intelligent wireless sensors with solar harvesting", in *Proc. of IEEE SSCI 2014 - 2014 IEEE Symposium Series on Computational Intelligence, IES 2014 - 2014 IEEE Symposium on Intelligent Embedded Systems*, 2014, pp. 1–7. DOI: 10.1109/INTELES.2014.7008978.

[7] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation", in *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011, pp. 164–174. DOI: 10.1145/2345156.1993518.

[8] V. Patil, Y. Mane, and S. Deshpande, "FPGA based power saving technique for sensor node in wireless sensor network (WSN)", *Studies in Computational Intelligence*, vol. 776, pp. 385–404, 2019. DOI: 10.1007/978-3-662-57277-1_16.

[9] V. Markevicius, D. Navikas, D. Andriukaitis, M. Cepenas, A. Valinevicius, M. Zilys, R. Malekian, A. Janeliauskas, W. Walendziuk, A. Idzkowski, "Two thermocouples low power wireless sensors network", AEU - International Journal of Electronics and Communications, vol. 84, pp. 242–250, 2018. DOI: 10.1016/j.aeue.2017.11.032.

[10] M. Tahir and K. Javed, *ARM® Microprocessor Systems: Cortex®-M Architecture, Programming, and Interfacing*. CRC Press, 2017.

[11] S. Cheshire and M. Baker, "Consistent overhead byte stuffing", *Computer Communication Review*, vol. 27, pp. 209–220, 1997.